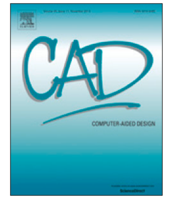
Computer-Aided Design 146 (2022) 103189

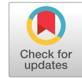
Contents lists available at ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad



Sweep Encoding: Serializing Space Subdivision Schemes for Optimal Slicing



M. Comino Trinidad [a,*], A. Vinacua [b], A. Carruesco [c], A. Chica [b], P. Brunet [b]

[a] *Universidad Rey Juan Carlos, Spain*
[b] *Universitat Politècnica de Catalunya, Spain*
[c] *HP Printing And Computing Solutions SLU, Spain*

A R T I C L E   I N F O

Article history:
Received 3 June 2021
Received in revised form 28 October 2021
Accepted 21 December 2021

Keywords:
Model slicing
3D printing
Hierarchical space subdivision
Octree

A B S T R A C T

Slicing a model (computing thin slices of a geometric or volumetric model with a sweeping plane) is necessary for several applications ranging from 3D printing to medical imaging. This paper introduces a technique designed to compute these slices efficiently, even for huge and complex models. We voxelize the volume of the model at a required resolution and show how to encode this voxelization in an out-of-core octree using a novel Sweep Encoding linearization. This approach allows for efficient slicing with bounded cost per slice. We discuss specific applications, including 3D printing, and compare these octrees' performance against the standard representations in the literature.

© 2022 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Slicing a geometric (surface-based or volume) 3D model is the operation that computes a complete set of thin or planar sections while a sweeping plane moves across the model.

This article introduces a technique designed to efficiently deal with the slicing process and overcome the main problems of classical approaches. These include the massive size of the intermediate representations when complex models are considered.

Slicing is an essential operation in several applications, including re-slicing processed 3D medical data to stacks of images across arbitrary directions, layer-based analysis of atmospheric and geological volume models, and 3D printing solid models, among others.

In our approach, instead of using the contour generation of standard 2D slicing algorithms, we voxelize the model space and use a volumetric scheme.

We differentiate between 2D slicing of (usually BRep) 3D models and volumetric slicing. While 2D slicing algorithms compute a sequence of intersections between a geometric model and a sweeping plane, volumetric slicing algorithms compute the sequence of slices—one voxel thick—generated by this sweeping plane. Classical 2D slices are computed by an object-plane intersection algorithm, whereas "boundary" pixels in volumetric slices correspond to voxels being stabbed by any portion of the object surface and are detected through a cubic cell—object surface intersection test. Both being essentially different, the results will also differ in many cases like objects having faces parallel to the sweeping plane. Nonetheless, applications using volumetric data like 3D medical data or 3D printing require volumetric slicing schemes to precisely model the thickness of the real-world slices and the 3D nature of the printer ink dots.

We also propose a novel serialization of space subdivision scheme, specifically well-suited for our problem. Hierarchical space subdivision solutions have been widely used for solid representation as they are easy-to-use data structures that offer many advantages. They allow significant compression of volume information, locally adapt to the model's features frequency, support robust boolean operations and morphological operators, and can be naturally linearized for off-core storage. In what follows, we will focus on octree representations.

Octrees require setting a maximum depth (resolution) at which to stop the recursive subdivision. This depth represents the precision the given octree can represent underlying models or the highest frequency of features it can capture. Currently, applications require higher and higher definitions, giving rise to very large octrees that, therefore, must be stored off-core and paged-in as required by the algorithms processing them. However, off-core storage is slower, and, consequently, we must pay attention to how we store the octree off-core to avoid hindering the performance of the algorithms of interest. Current commercial printers (like HP Jet Fusion 5200 Series) feature high resolutions of up to 1200 vpi in the xy plane and 300 slices per inch with printing beds of 12 × 16 × 16 inches, which puts severe demands on these algorithms. Each of the 4800 slices consists of roughly

* Corresponding author.
E-mail addresses: marc.comino@urjc.es (M. Comino Trinidad), alvar@cs.upc.edu (A. Vinacua), alex.carruesco.llorens@hp.com (A. Carruesco), achica@cs.upc.edu (A. Chica), pere@cs.upc.edu (P. Brunet).

https://doi.org/10.1016/j.cad.2021.103189
0010-4485/© 2022 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

2.7×10^8 voxels, and they must be produced at the advancement rate of the printer. Depending on the printing mode, each slice must be produced in less than seven seconds. This time must include some expensive computations to achieve a good printed part quality. Hence, in the printers we used (HP Jet Fusion Series), the slicing of the model must be done in less than 3 s.

Additionally, it is possible to apply different inks or treatments to the printed object in some machines. These are useful to change characteristics like colour and finish. As they add to the final cost and are only needed in the object surface, it is essential to identify and compute all voxels containing the object boundary, which we call grey voxels. However, 2D slicing is usually computed by intersecting the input model by a sweeping plane, which does not detect all grey voxels correctly (e.g., when the boundary of the part contains horizontal faces). Instead, we intersect the solid model with the cubical cells of the volume voxelization. This way, we can guarantee that grey voxels separate white from black voxels, and surface treatments are applied to the whole surface of the model. Volumetric processing also supports more precise computations of the voxel coverage along the boundary, further enhancing the achievable surface quality.

In this paper, we discuss a linearization of octrees especially conceived to make sequential volumetric slicing efficient. We dub this linearization *Sweep Encoding*, firstly introduced in patents [1,2]. We present algorithms to build such octrees and efficiently extract slices sweeping through the volume with bounded cost per slice, which is essential to guarantee the fast-enough computation of the slices during printing. The complexity of most slicing algorithms is input-data bounded (i.e., depends on the triangular mesh size), making it hard to ascertain hard bounds on the execution time required to compute a slice. Instead, our volumetric slicing algorithm is output-data bounded (i.e., depends on the printing resolution). Therefore, for a given application, it is possible to offer such performance guarantees, which—as we mentioned—are of paramount relevance in some applications. Finally, we also discuss a specific application of these Sweep octrees in 3D printing and compare this with the standard representations in the literature.

The rest of the paper is organized as follows. Section 2 reviews the most relevant previous work on the subject. In Section 3 we introduce our novel Sweep encoding and explain how it overcomes the limitations of existing ones. Section 4 explains how to construct our octrees from triangular meshes (top-down) and stacks of vectorial slices (bottom-up), and Section 5 describes how to slice them efficiently. Finally, Section 6 discusses the performance of our method against Depth and Breadth-first encodings, and, finally, concluding remarks are provided in Section 7.

2. Previous work

Slicing is commonly performed by obtaining a set of contours for each slicing plane. Since the planes go through model space, most 2D slicing algorithms simulate the movement of a sweeping plane throughout the mesh of the model [3]. Optimally, a slicing algorithm should scale with the number of intersecting triangles in each slice. The algorithm of Huang et al. [4] is optimal in this sense for a uniform distribution of slicing planes along their perpendicular axis.

It must be observed that any 2D slicing process involves two main intrinsic problems: the staircase effect and the containment problem. The first one is connected to the selected resolution and may be alleviated by applying adaptive slicing. This optimization can be done for triangle meshes [5] or directly to a surface model if it is available [6]. Minetto et al. [7] proposed an optimal algorithm for uniform and adaptive 2D slicing. Octrees can also be helpful to compute an adaptive slicing of a given model [8]. Given the total number of slices, positioning them in a provably optimal way is possible when using variable thickness [9]. Another way of attacking the staircase problem is to take advantage of the thermal diffusion of the deposition material, or to exploit free form material deposition [10], but this is only possible on Fused Filament printers equipped with a 6-DOF arm. The second issue (the containment problem) appears when slices do not stab the object surface and fall entirely inside or outside the model. Slices above the top of the object or below its bottom fall into this category, being completely outside in the case of slicing standard solid parts, but being totally interior, for instance, if what is sliced is the part mould. Algorithms must detect these cases and classify (in, out) the corresponding slices in a correct way, based for instance on neighbour slides, to avoid ending up with erroneous slices.

One important application for slicing is in additive manufacturing [11]. Slicing is one of the main challenges that need to be solved in 3D printing. Oropallo et al. [12] introduce a technique designed to deal with the slicing process efficiently. Given the resolutions needed for high-quality 3D printing, the discretization of the volume into black, grey, or white voxels needs to be performed efficiently. Spatial data structures are a cornerstone of most geometry processing algorithms [13,14]. For our problem, since we want to store the voxelization of a model to 3D print, it makes sense to use an octree [15]. At maximum subdivision, the octree perfectly reflects the uniform voxelization we want to represent while efficiently exploiting the variable detail inherent in the model. Indeed, octrees have been applied because of these same properties to many applications. From isosurface extraction [16,17] to surface reconstruction [18], segmentation [19], simplification [20], global illumination [21], and even neural networks [22]. Another consequence of the size of voxelizations for 3D printing is that they cannot fit into in-core memory even in octree form. Octrees have also been adapted to work out-of-core in many applications, like simplification [23], surface reconstruction [24], view-dependent visualization [25], and robotics [26]. For certain types of models, like large lattice structures, out-of-core efficient solutions can be devised without the use of octrees [27]. Instead, when comparing with algorithms that compute geometry slices, the cost of rasterizing the high-resolution slices also needs to be factored in for these applications.

Another critical issue is the size of the discretized volumes that are generated for current printers. Since current technology allows printing on the micrometre scale, the cost of storing the resulting voxelization can be prohibitive. Some compression is necessary. DICOM [28], for example, uses JPEG 2000 to compress models encoded in this format, but there are multiple methods available [29]. Modern approaches can also be based on learning the optimal quantization using deep neural networks [30]. However, only those that perform lossless compression are useful for printing [31]. These approaches optimize the applied compression algorithm to allow efficient rendering. Moreover, in the case of 3D printing, requirements also include fast and time-bounded slicing algorithms.

Our method combines all these properties into a single data structure. The octree allows us to perform voxelization efficiently in an out-of-core friendly way. The result is a compressed version of the voxelized model that can be read from disk slice by slice. This is perhaps its most important feature, as the slicing needs to be performed fast enough to avoid introducing deformations during printing. The octree also allows us to compute and encode all grey voxels efficiently. Thus, it can be seen as a compact representation of an occupancy grid, where each voxel is classified as either white, grey, or black. Black and white voxels are either entirely inside the solid or entirely outside, respectively. Grey voxels intersect the boundary and can carry any additional data that the printing process needs for surface treatment.

3. Linear octrees and traversal encodings

Octrees are well-known efficient representations for 3D objects. They are intrinsically a multiresolution representation, containing all members of the family of increasing object resolutions up to some maximum level of space subdivision [32]. The root node is assumed to be at the top of the octal tree, representing the contents of a box-shaped universe U. Region octrees [33] represent objects and 3D structures O contained in U by a hierarchical set of nodes N, each one representing a box-shaped region of U resulting from the iterated subdivision of the Universe. Region octree nodes encode their region's properties $B(N)$ in a predefined number of bits. The minimum number of bits per node b_N for representing homogeneous solid objects is 2, with the property taking the values *EMPTY*, *PARTIAL* and *FULL* [32].

The original octree representations used a tree structure using pointers [33]. However, as the complexity of the object shapes and their required resolution increased, the total octree size became a critical issue. Thus, several pointerless data structures were proposed, as the linear quadtrees and octrees with locational codes for non-empty nodes [34] and the depth-first node traversal sequential encodings [35]. Both proposals can be considered linear octree encodings, the tree being represented as a sequential list of nodes.

Traversal octree encodings are a particular case of linear octree encodings. They derive from different traversal algorithms, and they represent the tree as a long list of nodes with no pointers and no extra information. The final octree file size is $b_N * |N|$, where b_N is the number of bits per node and $|N|$ is the total octree nodes number. Given a traversal tree algorithm, its corresponding traversal octree encoding is the sequential list of nodes that results from this traversal. They are application-dependent, being efficient in applications requiring the encoded traversal.

Having application-dependent traversal octree encodings can be a wise solution. It is a compact scheme, and huge octrees can be sequentially requested from external memory in a very efficient way. Standard traversal octree encodings include Depth-first [35] and Breadth-first [13]. In Depth-first encodings, any node N_i is immediately followed in the list by the sub-trees of its sons, ordered according to a certain function $SortSons(N)$. In Breadth-first traversal lists, given any two nodes N_1 and N_2, if $N_1.depth < N_2.depth$, then N_1 precedes N_2. The function $SortSons(N)$ is application-dependent and defines the ordering among siblings. In rendering applications, for instance, it could impose a far-to-near node sorting.

In what follows, we discuss the main properties and applications of a different traversal octree encoding: the Sweep traversal encoding. Given a direction $\mathbf{d}$, let us consider the bundle of planes $P_\mathbf{d}$ perpendicular to $\mathbf{d}$ that sweep the universe U. In the Sweep encoding, nodes will be listed in the same order they get stabbed as $P_\mathbf{d}$ sweeps U.

We will basically restrict ourselves to the case in which $\mathbf{d}$ defines one of the three orthogonal directions of the U box. Without loss of generality, let us assume that $\mathbf{d} = \mathbf{z}_+$. Then, let N_a and N_b be two octree nodes. We denote by N_a^z the minimum z coordinate in node N_a. Then N_a will be listed before N_b if $N_a^z < N_b^z$ or if $N_a^z = N_b^z$ and $N_a.depth < N_b.depth$ or if $N_a^z = N_b^z$ and $N_a.depth = N_b.depth$ but $Morton(N_a^x, N_a^y) < Morton(N_b^x, N_b^y)$, where $Morton(a, b)$ is a function that computes the Morton code (see [36]) for a and b (it is worth noting that *Morton* is an arbitrary example of the aforementioned *SortSons* function, which is often used in linearizing 2D information [37,38]). Nodes in this Sweep encoding will be listed in appearance order when traversing the tree in a Depth-first way in z and in a Breadth-first way in the remaining two directions (x, y). See Fig. 1 for an example.

Using location codes for non-empty nodes [34] can be helpful in some applications that only require operating with *FULL*

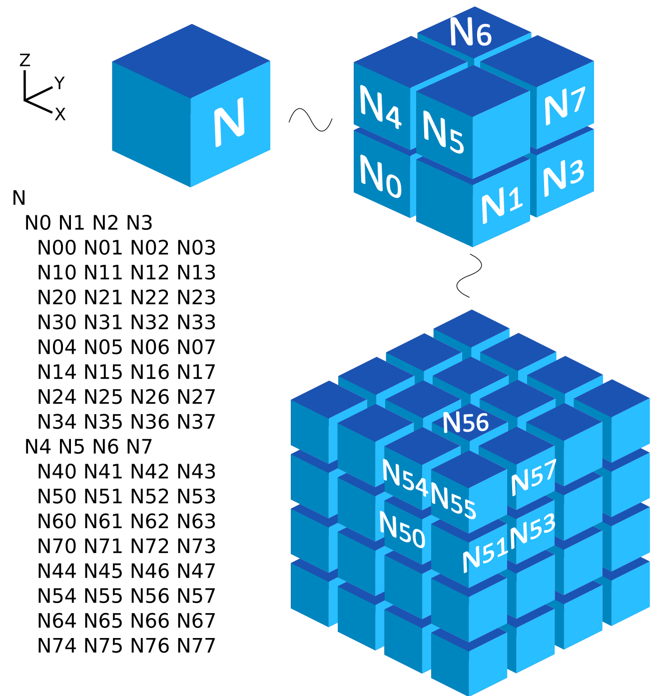


Fig. 1. Example of the Sweep Octree serialization for a simple octree of 3 depth levels.

nodes. However, its storage requirements are larger than Depth-first, Breadth-first, and Sweep encodings. Moreover, the particular interest of these three traversal encodings, mainly for out-of-core huge models, is application-dependent. The Breadth-first option can be well suited for progressive model refinement, such as model transmission to distant users. In contrast, Depth-first encodings have been successfully used in rendering algorithms. However, as shown in the following Sections, Sweep encoding outperforms Depth-first and Breadth-first in the case of sequentially slicing a 3D model by avoiding going back and forth in the out-of-core file.

4. Overview and sweep octree creation

When considering a representation strategy for 3D models to be sliced, we need to consider its construction, storage, and slicing costs. The model size (i.e., number of triangles) may directly impact the construction cost, not necessarily the storage cost. For instance, highly-tessellated flat surfaces will not translate into more complex octrees. Instead, the model complexity can directly impact both construction time and storage size. As the surface's geometric complexity increases, this translates into more heterogeneous filled/empty regions causing finer sub-division around the surface and larger octree sizes.

Here, we address the case where, while construction time and storage size are important, the most critical constraint is related to the slicing time. For example, each model slice takes a particular technology-dependent time t to get printed. This process must be continuous and cannot be delayed or interrupted. Printers rely on chemical and physical processes to solidify the slices. A delay in adding the next slice can distort the shape or the printed part's mechanical properties. Thus, each model slice must be available when the previous finishes printing, which meaning the generation of a slice cannot take longer than a given parameter t_{max}.

The 3D printers we have today come in various sizes and resolutions, but both tend to increase. A commercial printer may

have a bed of more than 15 inches on its longest side, with a resolution of 1200 vpi. The rasterization of a typical large format printer, stored most compactly, with only one bit per pixel, may occupy in the order of half a terabyte. Even applying image compression algorithms, the resulting voxelization would need many GB of data. On the other hand, that same voxelization can be represented with an octree of 15 levels, using a 2-bit encoding for each node to determine if it is interior, exterior, or contains the surface. We store each node using a 16-bit integer encoding the type of each of its 8 children. In real-world scenarios, the space to represent typically contains large homogeneous volumes (either solid or empty), so using octrees can dramatically reduce the memory required to encode these regions, as presented later in the results section. Compression techniques could further reduce the memory consumed by the octree, but decompression times would also need to be within t_{max}.

Our proposed Sweep encoding is optimized for slice retrieval in a specific sweep direction **d**. Without loss of generality, we will assume that the space has been rotated to make the z axis coincide with **d**. The user must also specify the rotation around z and the bounding box U containing the models (each represented by its octree root). In the previous example, the edge of this bounding box would be divided into 2^{15} voxels).

Slicing out-of-core Sweep octrees consist of obtaining the complete set of 2D planar sections at given positions when a sweeping plane moves across the model. Let us note S_z the slice defined by a specific z value, and N_{minZ} and N_{maxZ} the minimum and maximum z coordinates of a particular node N. In this context, Sweep-encoded models are slicing-friendly because of three main properties:

- The out-of-core file is sequentially traversed only once, with any octree node being accessed a single time. Slicing works by advancing a front through the out-of-core file.
- While any S_z is computed, only the subset of nodes fulfilling $N_{minZ} \leq z \leq N_{maxZ}$ must be stored in core memory. These are the active nodes regarding S_z, which are the subset AN that is used for the computation of S_z, see Section 5. As z is incremented, the front moves forward, reading new out-of-core nodes that become active nodes. Also, previously active nodes not fulfilling the updated inequality $N_{minZ} \leq z \leq N_{maxZ}$ are discarded and deleted from core memory (Section 5).
- The footprint of the active nodes in core memory is of the order of the length of the object surface in the slice. This is a direct consequence of the Quadtree Complexity Theorem together with its Octree generalization [13].

In short, the sequential slicing of out-of-core Sweep encoded models is optimal because out-of-core nodes are visited only once (we assume that fetching the data is the highest cost in the process). Due to the nature of the encoding, nodes are always sequentially accessed. Thus the encoding is also optimal in terms of page misses because misses will only occur when reading new memory blocks from the persistent storage. Hence, it is optimal in the sense that each node will be read from disk exactly once. Moreover, the slicing algorithm's footprint in core memory is bounded by the slices' geometric complexity. By measuring the cost per slice as the number of required out-of-core fetches, we can conclude that the average cost per slice is bounded and equals the total number of octree nodes divided by the number of slices.

Typical model inputs are either manifold triangular meshes or a stack of vectorial slices. For printing, these models must be discretized into regular empty/full cells, but a top-down or bottom-up strategy should be used depending on the type of model. These strategies are discussed next.

4.1. Top-down construction

The top-down construction approach is the straightforward method for octree construction when dealing with manifold triangular meshes. We recursively subdivide the Universe U into eight regular regions and test whether any mesh triangle intersects each of these. For such a task, we use a Fast Triangle-Box intersection test, based on the Separating Axis Theorem (see [39] and references therein). Each octree node is tested against the list of triangles that were found to intersect its parent. If an intersection is found, the node is labelled as intermediate grey and recursively subdivided until the target depth is reached. If no intersection is found, the node is classified as white (outside the mesh) or black (inside the mesh).

A simple test classifies nodes into white and black (exterior or interior). We cast a ray from the node's centre towards a random direction and find all the intersections with the mesh. If the number of intersections is odd, the node is classified as black (interior) and, otherwise, as white. In our implementation, this process is carried out with the help of the Embree [40] library.

Nodes are serialized and stored into disk at the same time they are traversed. The sweep traversal is implemented using a stack of queues. Each node is processed by testing whether any of its children intersect any mesh triangle. The ones that test positively (intermediate greys) are annotated to be recursively processed by queuing them into two different queues: one for the nodes with higher z coordinate and one for the nodes with lower z coordinate. Next, both queues are added to a stack. Nodes are processed in the order they appear in the queues inside the stack, effectively achieving the desired traversal.

4.2. Bottom-up construction

When dealing with stacks of polygonal slices, the top-down construction is a highly resource-demanding approach because a large number of polygons must be analysed to determine the nature of the octree nodes close to the root. Therefore, we propose a more efficient alternative method to process stacks of polygonal slices, which consists of computing the corresponding quadtrees for the slices while incrementally building an octree by collapsing consecutive quadtree nodes into octree nodes.

The algorithm iterates over the input polygonal slices sequentially, generating a single quadtree for all the polygons at the same height in the stack; this quadtree has the same size as the octree independently of the size of the polygons of the slice to ease the subsequent octree construction from the stack of quadtrees. Then, a pair of quadtrees can be merged into a grid of octrees because, for each of the 4 terminal nodes of a quadtree, we can find their adjacent 4 nodes in the other quadtree (subdividing a bigger terminal node when needed). The merge of these two quadtrees provides a set of small sub-octrees, which following the same criteria can be merged with the octrees from the resulting merge of the 2 subsequent quadtrees, generating bigger sub-octrees. This process is iterated, generating quadtrees and collapsing them in sub-octrees until all the polygonal slices have been processed. At this point, the final octree is generated from all the sub-octrees that have been incrementally built.

This approach is efficient in terms of algorithm complexity. Let E be the number of edges of the input, d the depth of the octrees and quadtrees. Note that the number of leaves of a fully subdivided tree is c^d, where c is the number of children at each node. Thus quadtrees and octrees may have up to 4^d and 8^d leaves, respectively, and we can state that $d = logN$, were N is the number of octree leaves. During the quadtrees generation, each edge will be visited and classified as many times as the depth of the tree, leading to a cost of $O(E\ logN)$. The step of combining

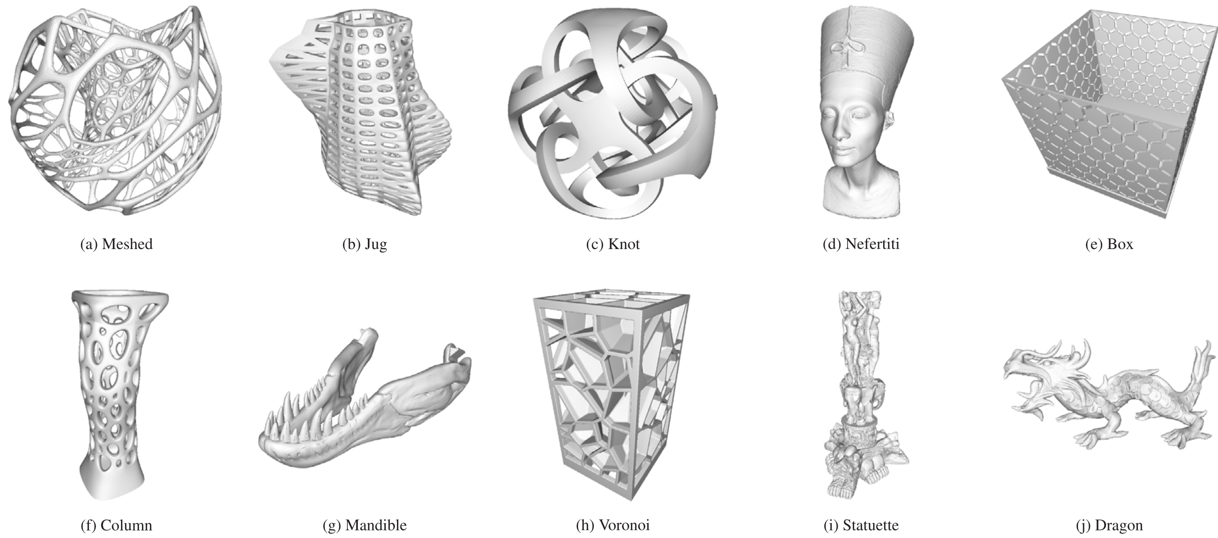


Fig. 2. Test models.

quadtrees to generate small octrees requires visiting the leaves of each quadtree once, and the root nodes of the resulting octrees are consequently visited to merge them into bigger octrees until reaching the root. Hence, the algorithm visits all the nodes of the final octree, and it is well known [13] that an octree with n non-terminal nodes has $N = 7\,n + 1$ leaves, so the cost of this step is $O(N)$. Finally, the total cost of the algorithm is the sum of both steps $O(N + E\ logN)$.

Nonetheless, octrees with moderate depths may potentially require a considerable amount of RAM to keep all the intermediate quadtrees and sub-octrees. Further optimizations were implemented by dumping intermediate sub-octrees to disk when they reach a certain depth. These are merged into a single octree once all the polygonal slices have been processed.

5. Octree-based sequential slicing

The sequential slicing operation can be defined as the acquisition of consecutive 2D slices, from a 3D representation, following a fixed direction. In the octrees domain, this can be understood as generating a sequence of quadtrees from an octree. As already mentioned, let us assume that the sequential slicing is performed towards the $+z$ axis. Our Sweep octrees allow us to rapidly obtain consecutive slices by maintaining a *state* of the slicing process, which is updated during the retrieval of each slice. Note that thanks to the Sweep octree serialization, this *state* requires a relatively small amount of in-core memory.

Let $N_i = (O, d)$ be a node of the octree, where O is the origin in 3D (the point of the node with minimal x, y, and z coordinates), and d is its depth level within the octree. Note that octree nodes are cubic, and their depth determines their size (i.e., $side(N_i) = 2^d$, assuming that the root node has the maximum depth and the leaves have 0 depth and unitary sides). Let AN be the list of N_i that intersect a given slice coordinate z. We will now show how to compute AN efficiently.

Let StQ be a stack of octree node queues, where each queue Q contains only nodes of a given depth in the octree. Before executing the algorithm, StQ is initialized with a queue containing only the root node. The algorithm iteratively processes the queue on the top of StQ until the stack is empty. At this point, all the nodes in AN can be used to produce the quadtree.

For each node N in the Q popped from StQ, we read the 16-bits integer codifying the types of its 8 children. If N intersects with z, we extract the 8 bits corresponding to the intersected children

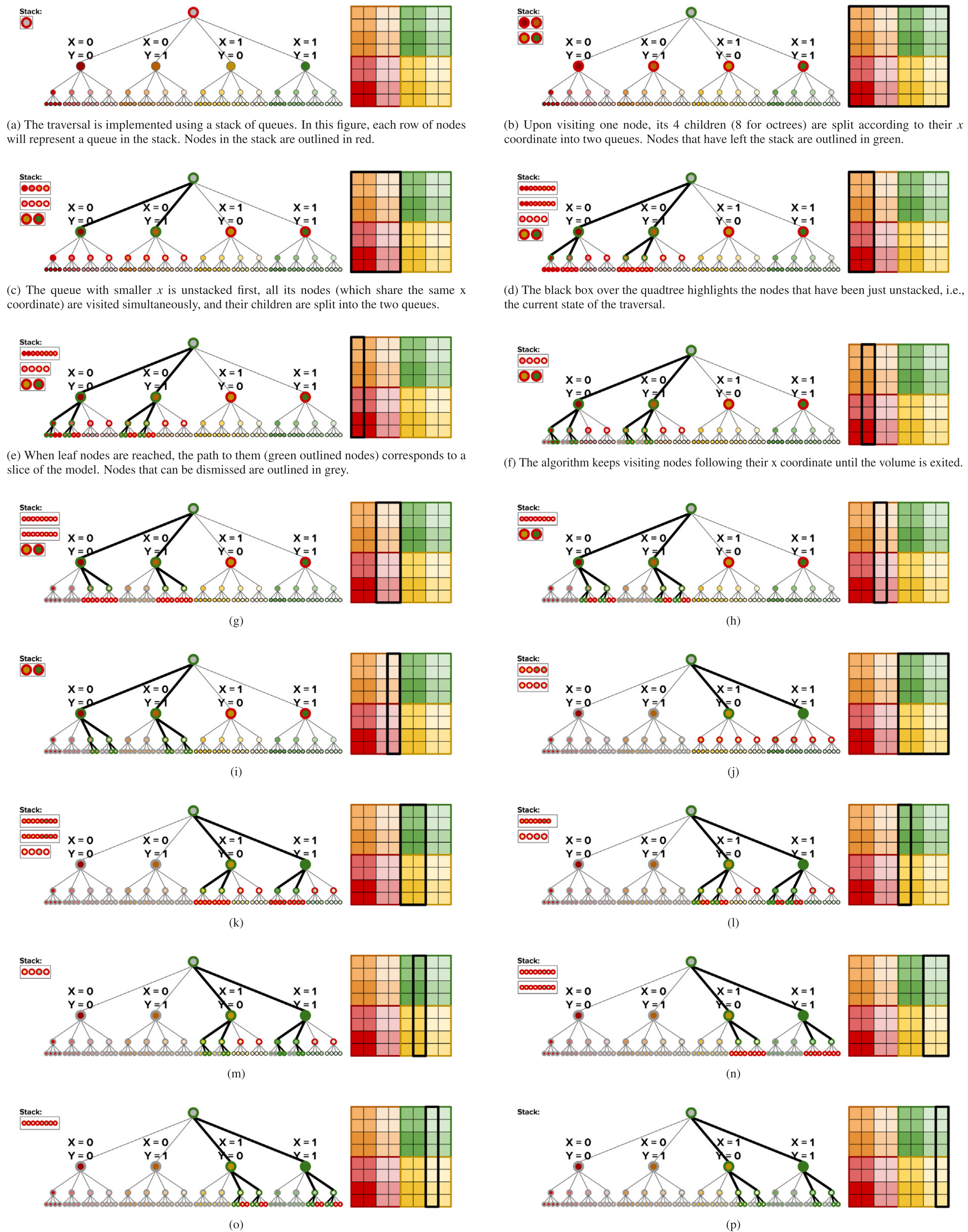
Algorithm 1 Sweep octree Slicing Algorithm

1: **procedure** GetSlice($root$, $sliceZ$)
2:   **if** $NotInitialized()$ **then**
3:     $Q \leftarrow Queue().Push(ROOT)$
4:     $StQ \leftarrow Stack < Queue > ().Push(Q)$
5:     $AN \leftarrow [\ ]$
6:   **else**
7:     $AN \leftarrow [N_i\ from\ AN\ if\ Intersects(N_i, sliceZ)]$
8:   **while** $StQ.NotEmpty()$ **do**
9:     $higherZ \leftarrow Queue()$
10:     $lowerZ \leftarrow Queue()$
11:     $Q \leftarrow StQ.TopAndPop()$
12:     **while** $Q.NotEmpty()$ **do**
13:       $N \leftarrow Q.FrontAndPop()$
14:       **if** $Intersects(N, sliceZ)$ **then**
15:         $C \leftarrow IntersectedChildren(N, sliceZ)$
16:         $AN.Add(C)$
17:       $higherZ.Push(N.HigherZGreyChildren())$
18:       $lowerZ.Push(N.LowerZGreyChildren())$
19:     **if** $higherZ.NotEmpty()$ **then**
20:       $StQ.Push(higherZ)$
21:     **if** $lowerZ.NotEmpty()$ **then**
22:       $StQ.Push(lowerZ)$

and add them to AN. Moreover, if any of the 4 descendants with lower z value is grey, these are added to the $lowerZ$ queue. Similarly, for those with higher z.

After all nodes in Q have been processed, the $higherZ$ queue is first pushed in StQ, followed by $lowerZ$. Thus, the algorithm will always visit first the nodes with lower z. Once the first queue with leaves is processed, AN will contain all the nodes intersecting the plane at z. Hence, the quadtree can be easily generated from this list by projecting each N_i onto the quadtree plane.

AN is reviewed to retrieve the next slice $z + 1$, and all nodes that do not intersect its plane are removed. Next, the recursive process is repeated: the queue on the top of StQ is extracted and processed (adding more queues to the stack) until a queue with leaf nodes is processed. The quadtree is finally extracted from the updated AN list. If the execution is halted, AN, StQ and the reading pointer to the octree file comprise the *state* needed to resume the



(a) The traversal is implemented using a stack of queues. In this figure, each row of nodes will represent a queue in the stack. Nodes in the stack are outlined in red.

(b) Upon visiting one node, its 4 children (8 for octrees) are split according to their x coordinate into two queues. Nodes that have left the stack are outlined in green.

(c) The queue with smaller x is unstacked first, all its nodes (which share the same x coordinate) are visited simultaneously, and their children are split into the two queues.

(d) The black box over the quadtree highlights the nodes that have been just unstacked, i.e., the current state of the traversal.

(e) When leaf nodes are reached, the path to them (green outlined nodes) corresponds to a slice of the model. Nodes that can be dismissed are outlined in grey.

(f) The algorithm keeps visiting nodes following their x coordinate until the volume is exited.

(g) (h) (i) (j) (k) (l) (m) (n) (o) (p)

Fig. 3. Sweep traversal illustrated on a quadtree. The sweeping direction goes along the x axis.

traversal. In Fig. 3 we illustrate this traversal using a quadtree instead of an octree for simplicity.

It is worth mentioning that the slicing time is variable since some slices have to parse a larger portion of the serialized octree to reach the tree leaves. However, these slices' extra time can be amortized because the next slices will be obtained much faster.

6. Results and discussion

To test our encoding efficiency, we ran tests on a set of public models (shown in Fig. 2). All tests were run on a commodity PC equipped with an Intel Core i7-4790K CPU, 32 GB of RAM, 16 GB of swap memory, and a 2 TB Toshiba DT01ACA200 HDD running

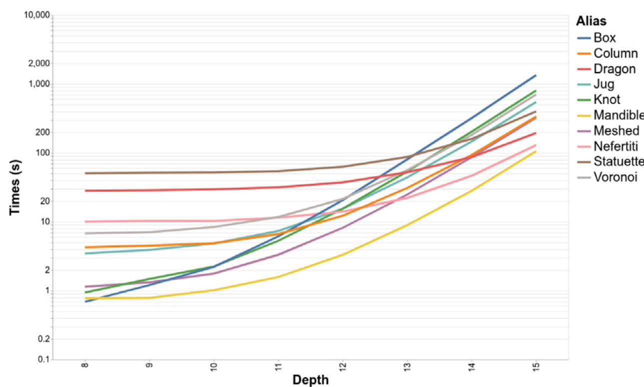


Fig. 4. Time in seconds to build each of the octrees of the models in our test set at different resolutions, plotted as a function of the depth of the octree. Notice the vertical scale is logarithmic.

Table 1
Times in seconds to build representative octrees of the models in our sample set. The heading indicates the depth of the octree.

	10	13	15
Meshed	1.76	24.77	326.51
Jug	4.83	44.03	546.71
Knot	2.24	53.77	804.15
Nefertiti	10.31	22.18	130.04
Box	2.21	80.52	1,345.52
Column	4.89	30.92	338.00
Mandible	1.01	8.95	106.16
Voronoi	8.40	56.18	704.72
Statuette	52.03	87.26	401.54
Dragon	29.57	52.30	195.20

Table 2
Memory (in MB) required to build representative octrees of the models in our sample set. The heading indicates the depth of the octree.

	10	13	15
Meshed	134	151	322
Jug	228	264	605
Knot	127	151	466
Nefertiti	611	663	703
Box	116	218	1,393
Column	299	315	528
Mandible	121	132	239
Voronoi	424	502	1,418
Statuette	2,414	2,412	2,663
Dragon	1,690	1,724	1,861

Table 3
Sizes of the test models.

Name	Vertices	Faces	Name	Vertices	Faces
Meshed	597K	1194K	Column	257K	516K
Jug	377K	754K	Mandible	79K	159K
Knot	69K	138K	Voronoi	1009K	2018K
Nefertiti	59K	118K	Statuette	5M	10M
Box	50K	103K	Dragon	14M	28M

Ubuntu 20.04. Table 3 gives basic data about the complexity of the chosen models.

However, our algorithm is not very sensitive to the input mesh's complexity (measured as triangle and vertex counts). Instead, it is more to the resolution at which we want to generate the raster slices and the frequency and relative positions of the surface's details.

Therefore, we compare the results obtained by constructing octrees of all the test models at different resolutions. Table 1 gives the elapsed times to build a Sweep octree (top-down) of each

Table 4
Storage in MB of the resulting octrees for the models in our sample set. The heading indicates the depth of the octree. We also provide storage spaces for the compressed octrees using an algorithm (ZIP) that allows sequential partial decompression.

	10	10 (Zip)	13	13 (Zip)	15	15 (Zip)
Meshed	1.49	0.71	96.37	37.12	1543.21	475.90
Jug	2.01	0.80	131.22	37.16	2103.80	451.01
Knot	3.79	1.29	243.76	63.07	3902.20	759.95
Nefertiti	0.46	0.15	29.33	8.19	469.53	118.45
Box	4.07	0.24	368.64	7.49	4292.74	57.41
Column	1.40	0.53	90.07	27.65	1441.63	367.97
Mandible	0.46	0.18	29.58	9.15	473.54	119.86
Voronoi	3.16	0.89	203.58	39.33	3258.46	463.31
Statuette	1.15	0.51	75.24	26.83	1205.51	363.29
Dragon	0.33	0.16	33.98	12.85	544.43	176.59

of these models at three significant depths. We focus on mid-sized to large octrees as the growth in complexity of the trees is exponential, and small trees are efficient without resorting to any smart techniques. Here, we have chosen the depths of 10, 13, and 15 to illustrate the proposed technique's behaviour. Depth 10 is a small to medium octree, but depth 15 can be considered large, with raster slices of about one terapixels. Notice that the construction times, all measured in seconds, are reasonable for most models and resolutions. While some models at high resolution may require a longer time, this is acceptable because we target applications where the slicing is the critical operation that needs to be fast. We have also implemented an octree encoded in depth-first order and another in breath-first order. We have observed that construction times for all three encodings are comparable, with depth and breadth strategies taking at most 5% less time for complex models.

In terms of resources needed, the memory footprint of the algorithms is also relevant. Table 2 gives the maximum memory allocated during the construction of each of the chosen test octrees in MB. One can see the variation depending on each model's complexity, but they are all suitable for current commodity PCs. The time and memory required to build the octrees are also shown in Figs. 4 and 5. In our Depth-first and Breadth-first octrees implementations, the depth-first strategy also shows maximum memory bounds down to a 30% lower than our encoding strategy. This happens because our strategy is partially a breadth-like traversal (that is, more nodes need to be allocated in the queues simultaneously). However, the memory consumed by the Breadth-first strategy is notably higher, making the algorithm crash during the 14-levels octree construction of some models (knot, box, Voronoi) and the 15-levels construction of all of them but two (Nefertiti, mandible). Moreover, none of the alternative encodings are any competition when we turn to slicing times.

Hard-disk storage space can also be relevant for some scenarios. Table 4 shows the resulting size for the chosen test octrees in MB. This table shows storage sizes for Sweep-encoded models, but similar sizes were obtained for breadth and depth ones. Octrees are already efficient space-saving structures. Our results show that, as one would expect, the octree size is more or less quadrupled every time we increase the subdivision depth (notice that the $\log_4$ of the storage size in bytes is roughly the depth of the octree). We also argue that further storage savings can be achieved by compressing these octrees using any sequentially coherent compression technique. While the compression step can take several minutes, this can be considered part of the preprocessing. During slicing, decompression can be performed much faster and can be limited to the part of the octree containing the slice. After compression, most storage sizes were reduced to between a half and a fourth of the original size.

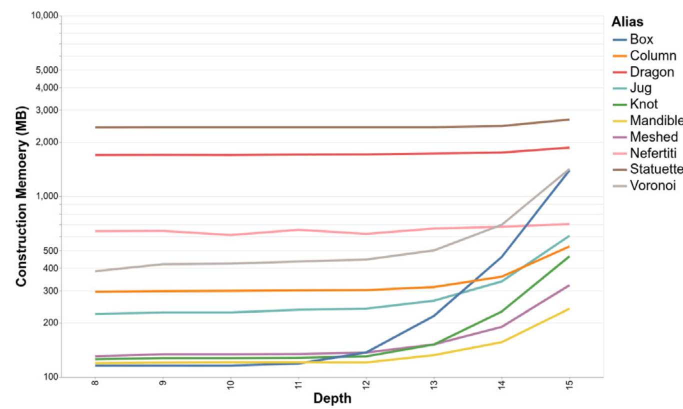


Fig. 5. Memory required to build the octrees for the different models in our test set, as a function of the desired maximum depth desired. Notice that the vertical axis has a logarithmic scale.

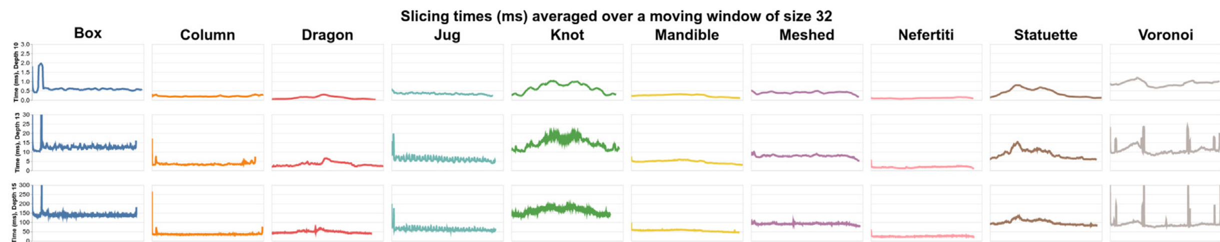


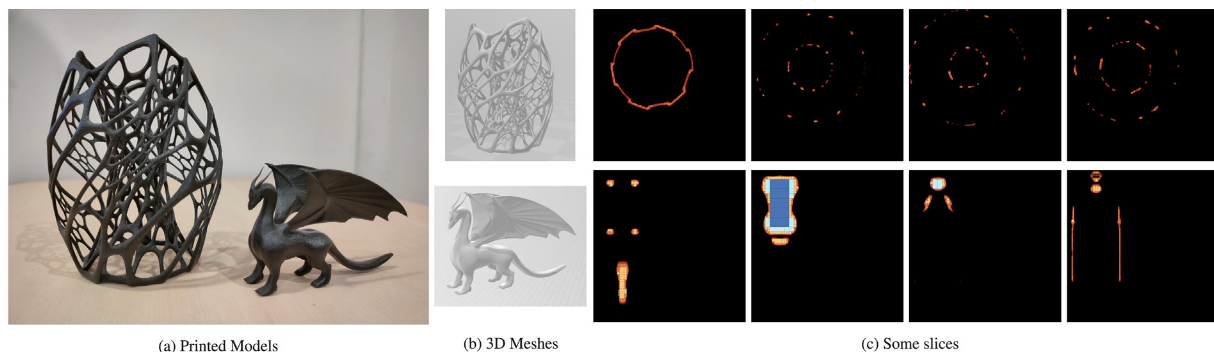
Fig. 6. From top to bottom, evolution of a moving average of the slicing times for octrees of depth 10, 13 and 15. By buffering a few slices, very steady and fast response times are achieved.

Table 5
Slicing times for our test cases at representative depths. “min.time” is the shortest time in seconds amongst all slices of the model, while “max” is the maximum for one slice. We also list mean and median as an indication of the distribution. The last column shows the maximum of a moving average with a window of size 32 over the slicing time. Even in cases where there is a high peak at some point, the moving average has a much smaller maximum, indicating these complex slices are sparse.

	dpth.	min.time	mean	median	max	max.amort(32)
Box	10	0.0003290000	0.0006783817	0.0005350000	0.0159600000	0.00196478
	13	0.00978800	0.01348064	0.01215200	1.0874420	0.11662597
	15	0.128302	0.143700	0.138041	15.270142	1.80276238
Column	10	0.0001270000	0.0002126273	0.0001850000	0.0020490000	0.00032816
	13	0.002403000	0.003557707	0.003351000	0.136020000	0.01723337
	15	0.03177600	0.03575966	0.03487100	2.10993000	0.26555287
Dragon	10	0.0000190000	0.0001137778	0.0000840000	0.0005380000	0.00029003
	13	0.00201400	0.00343126	0.00304900	0.00917500	0.00665094
	15	0.03717300	0.04803933	0.04586200	0.10158000	0.06629469
Jug	10	0.0000560000	0.0003455797	0.0003050000	0.0018240000	0.00061178
	13	0.002517000	0.006002136	0.005585000	0.079564000	0.01949256
	15	0.04162600	0.06113728	0.05892350	1.22153000	0.20002022
Knot	10	0.0000970000	0.0006414196	0.0006400000	0.0021130000	0.00102937
	13	0.00863800	0.01474788	0.01455550	0.02802500	0.01960109
	15	0.1257620	0.1619408	0.1624390	0.3329180	0.18454428
Mandible	10	0.0000500000	0.0002305523	0.0002155000	0.0005830000	0.00031966
	13	0.002720000	0.004725723	0.004714000	0.029925000	0.00754619
	15	0.04285900	0.05503441	0.05559300	0.46222400	0.09920759
Meshed	10	0.0000600000	0.0003918072	0.0003725000	0.0008790000	0.00050597
	13	0.004641000	0.007820344	0.007691000	0.013321000	0.01005569
	15	0.06898100	0.09168356	0.09104500	0.24655500	0.10723669
Nefertiti	10	0.0000160000	0.0001066303	0.0000990000	0.0006200000	0.00015656
	13	0.000776000	0.001838721	0.001762000	0.011899000	0.00596553
	15	0.01665000	0.02470449	0.02437200	0.08321300	0.05752359
Statuette	10	0.000067000	0.000374388	0.000297000	0.001422000	0.00079544
	13	0.004921000	0.008796875	0.008080500	0.020403000	0.01530506
	15	0.07598600	0.09907725	0.09499900	0.20520000	0.13177856
Voronoi	10	0.0003260000	0.0009024385	0.0007810000	0.0026080000	0.00119328
	13	0.00675600	0.01145959	0.01009000	0.15259100	0.02350075
	15	0.06658200	0.09016686	0.08421100	2.04826900	0.32094069

Table 6
Slicing times in seconds when the octree is stored in depth and breadth-first order. In these cases, each slice requires reading essentially the whole tree, so there is little variability between slices; therefore, we have not computed all slices, but just sampled 20 different slices across the volume of each example part. The results confirm this lack of variability, as witnessed by the small difference between minimum and maximum slicing time. Therefore, we do not include the mean and median, but list instead the difference between maximum and minimum time, and the relative increment from min to max. Notice that the relative increment may be large in cases where the slices are computed very fast (simple models and shallow octrees), but is consistently small when the task is more demanding.

		Depth-first encoding				Breadth-first encoding			
	dpth.	min.time	max	max–min	max–min(rel)	min.time	max	max–min	max–min(rel)
Box	10	0.1560960	0.1754920	1.94E−02	12.43%	0.178665	0.185229	6.56E−03	3.67%
	13	10.67690	10.73020	5.33E−02	0.50%	11.8314	11.9644	1.33E−01	1.12%
	15	174.7370	176.8460	2.11E+00	1.21%	–	–	–	–
Column	10	0.05390520	0.06253500	8.63E−03	16.01%	0.0628137	0.0668119	4.00E−03	6.37%
	13	3.533530	3.550640	1.71E−02	0.48%	4.00231	4.09356	9.13E−02	2.28%
	15	61.12040	61.30920	1.89E−01	0.31%	–	–	–	–
Dragon	10	0.01289600	0.01752450	4.63E−03	35.89%	0.0146542	0.0196814	5.03E−03	34.31%
	13	1.331730	1.363930	3.22E−02	2.42%	1.49522	1.55209	5.69E−02	3.80%
	15	25.05980	25.11160	5.18E−02	0.21%	–	–	–	–
Jug	10	0.07746020	0.08748280	1.00E−02	12.94%	0.0897739	0.0978147	8.04E−03	8.96%
	13	5.166720	5.204140	3.74E−02	0.72%	5.82232	5.87855	5.62E−02	0.97%
	15	98.62420	98.97470	3.50E−01	0.36%	–	–	–	–
Knot	10	0.1455730	0.1607500	1.52E−02	10.43%	0.169513	0.185029	1.55E−02	9.15%
	13	9.492730	9.535580	4.28E−02	0.45%	10.8792	11.0553	1.76E−01	1.62%
	15	171.1370	173.8530	2.72E+00	1.59%	–	–	–	–
Mandible	10	0.01784080	0.01925900	1.42E−03	7.95%	0.0202381	0.0233525	3.11E−03	15.39%
	13	1.164350	1.201720	3.74E−02	3.21%	1.28614	1.32016	3.40E−02	2.65%
	15	20.1627	20.3921	2.29E−01	1.14%	21.0691	42.1305	2.11E+01	99.96%
Meshed	10	0.05738960	0.06947490	1.21E−02	21.06%	0.0667329	0.0713881	4.66E−03	6.98%
	13	3.761970	3.804020	4.21E−02	1.12%	4.19503	4.23941	4.44E−02	1.06%
	15	71.21270	72.77700	1.56E+00	2.20%	–	–	–	–
Nefertiti	10	0.01755580	0.02362570	6.07E−03	34.57%	0.0200945	0.0241078	4.01E−03	19.97%
	13	1.160070	1.191690	3.16E−02	2.73%	1.28258	1.38695	1.04E−01	8.14%
	15	19.89330	19.97000	7.67E−02	0.39%	20.7297	33.2014	1.25E+01	60.16%
Statuette	10	0.0444256	0.0538408	9.42E−03	21.19%	0.0511554	0.0554931	4.34E−03	8.48%
	13	2.951350	2.982650	3.13E−02	1.06%	3.28848	3.39821	1.10E−01	3.34%
	15	55.77830	55.97120	1.93E−01	0.35%	–	–	–	–
Voronoi	10	0.1216560	0.1289320	7.28E−03	5.98%	0.141415	0.152172	1.08E−02	7.61%
	13	8.078500	8.121470	4.30E−02	0.53%	8.91562	9.08109	1.65E−01	1.86%
	15	154.474	157.397	2.92E+03	1.89%	–	–	–	–



(a) Printed Models (b) 3D Meshes (c) Some slices

Fig. 7. Examples of 3D-printed models, their corresponding 3D mesh and some of the generated slices at different z. In this case, the z axis is aligned along the vertical direction. Voxels are drawn with a 1 pixel black margin for visualization purposes, and a colour scale is used to depict their size (blue illustrates big voxels and red illustrates small voxels). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Next, we turn to the times spent to compute slices in our proposed encoding. We have computed equally spaced slices at the chosen resolution for each model in our test set for octrees of depths 10, 13, and 15. Table 5 shows for each of these how long it took to obtain the quickest (min.time column) and the slowest (max column) slices for the whole sweep. We also report the mean and median values of the times required for each of the slices. Notice that while for large complex models some slices take long (there is a slice of the Box model at the highest resolution that took 15 s to compute, although this is a rare event), this does not often happen, as shown by the small mean and median times in those same cases. Should an application require assurance that slices will be provided within a fixed time budget, one may buffer a few slices to achieve this. As an example, the last column of Table 5 shows the amortized time (averaged time over a batch) per slice if one builds a buffer of 32 slices; Fig. 6 shows the evolution of the moving average of the slicing times with a window of 32 slices.

The spikes in these plots correspond to outlier times for some slices, but the moving average is much smaller, showing that these outliers are sparse. Larger buffer sizes may be used in general to achieve even shorter amortized times.

We have computed the slicing times for the two most usual octree serialization schemes as a contrasting reference. Table 6

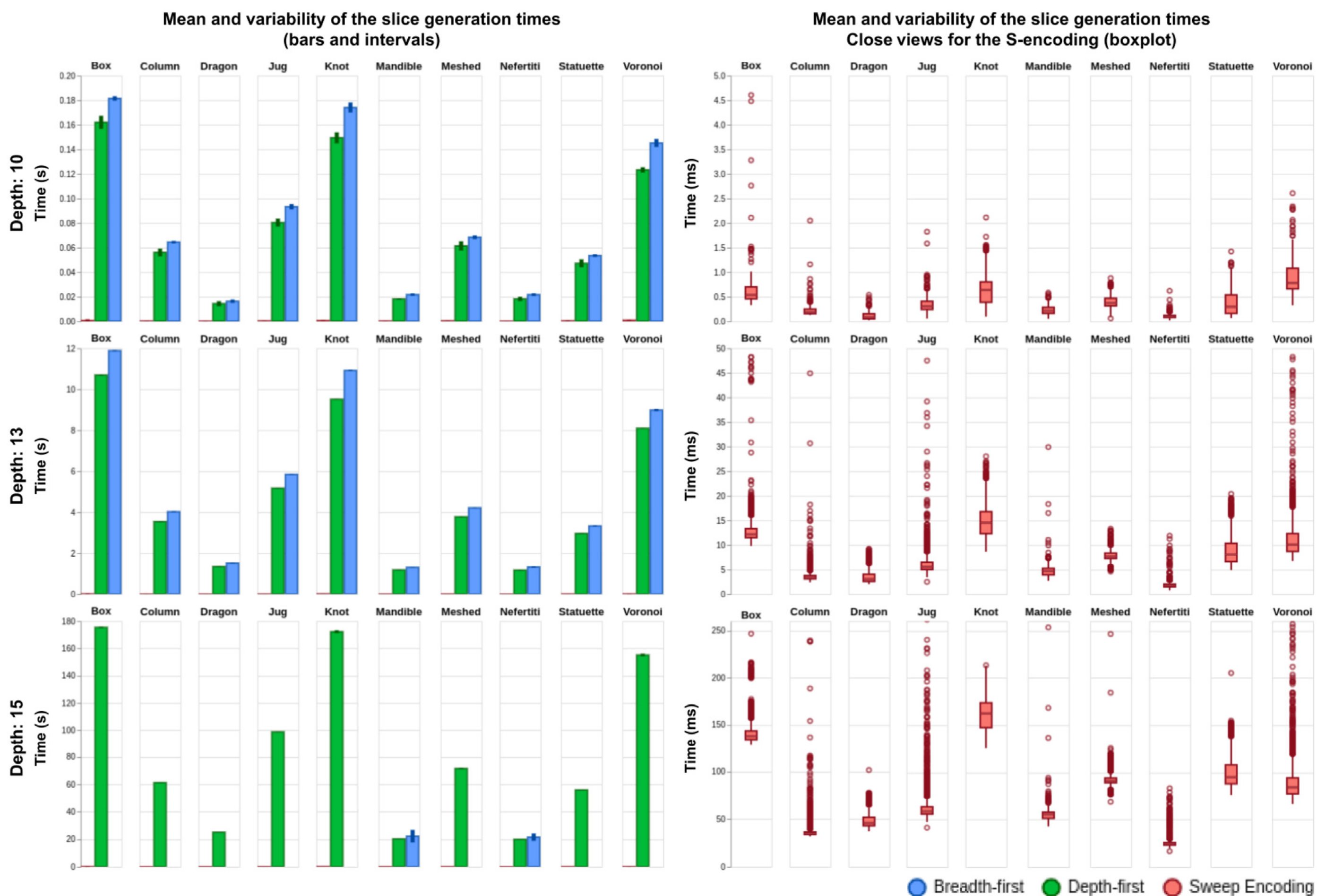


Fig. 8. Mean and variability of the slice generation times using our S-encoding or depth and breadth-like encodings. From top to bottom: slicing times for octrees of depth 10, 13, and 15. For the breadth-first encoding and depth 15, we were only able to produce slices for the Mandible and Nefertiti models.
The first column displays bars encoding the average slice time and small intervals showing their variances. However, slicing times for the S-encoding are much smaller and, at the chosen scale, their values look close to 0. Hence, the right column shows boxplots of the S-encoding slicing times at a much finer scale. These charts have been clipped to 5, 50 and 250 ms respectively, leaving out less than 0.1% of the values.
Our algorithm outperforms both BF and DF. This difference increases proportionally to the model complexity. Depth and breadth encodings present less relative variability than our method since because generating takes fixed time, equivalent to parsing the whole octree from disk. Instead, our algorithm has more outliers since the sequential slice generation time depends on the read octree chunk size. This can be alleviated by buffering strategies, as shown in Fig. 6.

gives the times to compute slices in a depth-first octree for the same models and depths. Here we only report the minimum and maximum time since these are relatively close in all cases. For convenience, we also included a column (labelled max–min) displaying the difference between times for the fastest and slowest slice in each case, as well as the relative difference. This serialization order makes it necessary to read the entire octree to compute each slice.

As a result, there is slight variation in slicing times, as the computation is dominated by the traversal cost, as shown in Fig. 8 Therefore, the numbers reported correspond not to an entire span of slices but to twenty slices that sample the whole span of the model in each case.

7. Conclusions

In this paper, we have introduced a novel scheme to compute 3D object slices efficiently, even for huge and complex models. The proposed algorithm can be useful in different applications, we included some 3D printing results in Fig. 7. We voxelize the volume of the model at a required resolution and show how to encode this voxelization in an out-of-core octree linearized in a Sweep Encoding that allows for efficient slicing with bound cost per slice.

We have concluded that slicing out-of-core Sweep-encoded models is optimal because out-of-core nodes are visited only once (we assume that this is the highest cost in the process). Moreover, we have seen that the slice geometric complexity bounds the slicing algorithm's core memory footprint. By measuring the cost per slice as the number of required out-of-core fetches, we can conclude that the average cost per slice is bounded.

We have also compared the proposed Sweep Octree encoding against the standard representations in the literature in terms of performance. We have shown that Sweep encoding outperforms them in several examples.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: The authors of the paper currently hold employment relationships with the Universitat Politécnica de Catalunya (www.upc.edu), with the Universitat de Vic (www.uvic.cat), with the Universidad Rey Juan Carlos (www.urjc.es) and with HP Printing And Computing Solutions SLU.

Acknowledgements

This work has been partially funded by the Spanish Ministry of Science and Innovation (MCIN / AEI / 10.13039/501100011033) and FEDER (‘‘A way to make Europe’’) under grant TIN2017-88515-C2-1-R.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cad.2021.103189.

References

[1] Rosello LA, Vinacua A, Brunet P, Comino M, ROGEL JG, Gonzalez S, et al. Serialising a representation of a three dimensional object. Google Patents; 2020, US Patent 10, 853, 998.
[2] Llorens AC, Vinacua A, Brunet P, Gonzalez S, Rogel JG, Comino M, et al. Sub-volume octrees. Google Patents; 2021, US Patent 11, 003, 166.
[3] McMains S, Séquin C. A coherent sweep plane slicer for layered manufacturing. In: Proceedings of the fifth ACM symposium on solid modeling and applications; 1999. p. 285–295.
[4] Huang X, Yao Y, Hu Q. Research on the rapid slicing algorithm for NC milling based on STL model. In: Asian simulation conference. Springer; 2012, p. 263–71.
[5] Pandey PM, Reddy NV, Dhande SG. Real time adaptive slicing for fused deposition modelling. Int J Mach Tools Manuf 2003;43(1):61–71.
[6] Ma W, But W-C, He P. NURBS-based adaptive slicing for efficient rapid prototyping. Comput Aided Des 2004;36(13):1309–25.
[7] Minetto R, Volpato N, Stolfi J, Gregori RM, Da Silva MV. An optimal algorithm for 3D triangle mesh slicing. Comput Aided Des 2017;92:1–10.
[8] Siraskar N, Paul R, Anand S. Adaptive slicing in additive manufacturing process using a modified boundary octree data structure. J Manuf Sci Eng 2015;137(1).
[9] Alexa M, Hildebrand K, Lefebvre S. Optimal discrete slicing. ACM Trans Graph 2017;36(1):1–16.
[10] Etienne J, Ray N, Panozzo D, Hornus S, Wang CC, Martínez J, et al. CurviSlicer: Slightly curved slicing for 3-axis printers. ACM Trans Graph 2019;38(4):1–11.
[11] Gao W, Zhang Y, Ramanujan D, Ramani K, Chen Y, Williams CB, et al. The status, challenges, and future of additive manufacturing in engineering. Comput Aided Des 2015;69:65–89.
[12] Oropallo W, Piegl LA. Ten challenges in 3D printing. Eng Comput 2016;32(1):135–48.
[13] Samet H. Foundations of multidimensional and metric data structures. Morgan Kaufmann; 2006.
[14] Mehta DP, Sahni S. Handbook of data structures and applications. Taylor & Francis; 2018.
[15] Chen HH, Huang TS. A survey of construction and manipulation of octrees. Comput Vis Graph Image Process 1988;43(3):409–31.
[16] Wilhelms J, Van Gelder A. Octrees for faster isosurface generation. ACM Trans Graph 1992;11(3):201–27.
[17] Ju T, Losasso F, Schaefer S, Warren J. Dual contouring of hermite data. In: Proceedings of the 29th annual conference on computer graphics and interactive techniques; 2002. p. 339–346.
[18] Kazhdan M, Hoppe H. Screened poisson surface reconstruction. ACM Trans Grap (ToG) 2013;32(3):1–13.
[19] Vo A-V, Truong-Hong L, Laefer DF, Bertolotto M. Octree-based region growing for point cloud segmentation. ISPRS J Photogramm Remote Sens 2015;104:88–100.
[20] Schaefer S, Warren J. Adaptive vertex clustering using octrees. In: SIAM geometric design and computing. Citeseer; 2003.
[21] Kämpe V, Sintorn E, Assarsson U. High resolution sparse voxel DAGs. ACM Trans Graph 2013;32(4):1–13.
[22] Wang P-S, Liu Y, Guo Y-X, Sun C-Y, Tong X. O-cnn: Octree-based convolutional neural networks for 3d shape analysis. ACM Trans Graph 2017;36(4):1–11.
[23] Cignoni P, Montani C, Rocchini C, Scopigno R. External memory management and simplification of huge meshes. IEEE Trans Vis Comput Graphics 2003;9(4):525–37.
[24] Bolitho M, Kazhdan M, Burns R, Hoppe H. Multilevel streaming for out-of-core surface reconstruction. In: Proceedings of the fifth Eurographics symposium on geometry processing. Eurographics Association; 2007, p. 69–78.
[25] Lindstrom P. Out-of-core construction and visualization of multiresolution surfaces. In: Proceedings of the 2003 symposium on interactive 3d graphics; 2003. p. 93–102.
[26] Elseberg J, Borrmann D, Nüchter A. One billion points in the cloud–an octree for efficient processing of 3D laser scans. ISPRS J Photogramm Remote Sens 2013;76:76–88.
[27] Liu S, Liu T, Zou Q, Wang W, Doubrovski EL, Wang CC. Memory-efficient modeling and slicing of large-scale adaptive lattice structures. 2021, arXiv preprint arXiv:2101.05031.
[28] DICOM Standards Committee. Dicom. 2020, Accessed: 2020-07-14, https://www.dicomstandard.org/.
[29] Rodriguez MB, Gobbetti E, Guitián JAI, Makhinya M, Marton F, Pajarola R, et al. A survey of compressed GPU-based direct volume rendering. In: Eurographics (STARs). 2013, p. 117–36.
[30] Tang D, Singh S, Chou PA, Hane C, Dou M, Fanello S, et al. Deep implicit volume compression. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition; 2020. p. 1293–303.
[31] Komma P, Fischer J, Duffner F, Bartz D. Lossless volume data compression schemes. In: SimVis. 2007, p. 169–82.
[32] Meagher D. Geometric modeling using octree encoding. Comput Graph Image Process 1982;19(1):129–47.
[33] Samet H, Webber R. Hierarchical data structures and algorithms for computer graphics. I. Fundamentals. IEEE Comput Graph Appl 1988;8(3):48–68.
[34] Gargantini I. An effective way to represent quadtrees. Commun ACM 1982;25(12):905–10.
[35] Kawaguchi E, Endo T. On a method of binary picture representation and its application to data compression. IEEE Trans Pattern Anal Mach Intell 1980;2(1):27–35.
[36] Lorton KP, Wise DS. Analyzing block locality in Morton-order and Morton-hybrid matrices. ACM SIGARCH Comput Archit News 2007;35(4):6–12.
[37] Huang C-Y, Chung K-L. Manipulating images by using run-length Morton codes. Int J Pattern Recognit Artif Intell 1997;11(06):889–907.
[38] Vinkler M, Bittner J, Havran V. Extended Morton codes for high performance bounding volume hierarchy construction. In: Proceedings of high performance graphics; 2017. p. 1–8.
[39] Akenine-Möller T. Fast 3D triangle-box overlap testing. In: ACM SIGGRAPH 2005 courses. SIGGRAPH ’05, New York, NY, USA: Association for Computing Machinery; 2005, http://dx.doi.org/10.1145/1198555.1198747, 8–es.
[40] Wald I, Woop S, Benthin C, Johnson GS, Ernst M. Embree: A kernel framework for efficient CPU ray tracing. ACM Trans Graph 2014;33(4). http://dx.doi.org/10.1145/2601097.2601199.